Tiny Application-Specific Programmable Processor for BCH Decoding

Anthony Van Herrewege
ESAT/COSIC, KU Leuven, Belgium
Email: *anthony.vanherrewege@esat.kuleuven.be*

Ingrid Verbauwhede
ESAT/COSIC, KU Leuven, Belgium
Email: *ingrid.verbauwhede@esat.kuleuven.be*

***Abstract*— We present a novel design for a tiny application-specific programmable processor for BCH decoding. The design is optimized for use in a PUF key extractor, where low-area overhead is extremely important. Due to it's flexible nature, it can support a wide range of BCH codes. The complete design for a BCH(413, 296, 13) decoder requires only 1% (less than 70 slices) of the available resources of a small FPGA.**

***Index Terms*—BCH decoding, processor design, PUF key extraction, FPGA design**

I. Introduction

One of the requirements of most cryptographic systems is the ability to securely generate, store and recover high-quality secret keys. The high-quality property requires the key to be both unique and unpredictable. The fact that generating such a secure key is not trivial was recently once again made clear by Lenstra et al. [1], who showed that a large amount of public RSA keys share the same prime factors, making them instantly exploitable. Designing secure storage for keys is not trivial either and often increases system implementation overhead.

Physically Unclonable Function (PUF) [2] key extractors [3–5] aim to solve both these problems. Each physical instantiation of an extractor produces a unique, unpredictable, fixed key by design, generated from the inherent randomness of the PUF. Since the key can always be regenerated with the extractor, there is no need for expensive, secure non-volatile memory. An essential part of any PUF key extractor is an error correction block.

Contribution: We present a novel design for a tiny and application-specific programmable processor for BCH decoding, a perfect fit for use in a PUF key extractor.

Paper outline: In Section II, we introduce the notation used throughout the paper and give background information on BCH code construction and decoding algorithms. Section III describes the design of our processor. Results for synthesis and runtime are presented in Section IV. Finally, conclusions are given in Section V.

II. Background

In this section, the notation used throughout the paper is explained. Next, we look at the mathematical background of BCH codes. A short overview of BCH code construction is given and the ideas behind BCH decoding algorithms are shown. Since this paper focuses on the design and implementation of a processor, we do not go deeper into the mathematics behind these algorithms.

A. Notation

A binary Galois field is written as $\mathbb{F}_{2^x}$. The symbol $\oplus$ is an addition over $\mathbb{F}_{2^x}$, i.e. a XOR operation, and $\otimes$ a multiplication. An element of $\mathbb{F}_{2^x}$ is written in capitals, e.g. A. $\mathcal{C}(n,k,t)$ stands for a BCH code with code length n, data length k and number of corrigible errors t.

B. BCH code construction

A BCH code $\mathcal{C}(n,k,t)$ is defined by its generator polynomial $\mathcal{G}$, which is constructed as follows [6, 7]. First, one selects the size u of the underlying field $\mathbb{F}_{2^u}$. Let $A \in \mathbb{F}_{2^u}$ be of order $\mathrm{ord}(A)$. For each $A^i, i = b, \ldots, b+2t-1$, define $\mathcal{M}_i$ as the minimal polynomial of A^i. $\mathcal{G}$ is defined as the least common multiple of all $\mathcal{M}_i$. This gives a code of length $n = \mathrm{ord}(A)$, with $k = n - \mathrm{ord}(\mathcal{G})$. In this paper, we only consider codes for which $A = \alpha$, a primitive element of $\mathbb{F}_{2^u}$, and $b = 1$, i.e. primitive narrow-sense BCH codes.

Codewords C are created by padding data word $D \in \mathbb{F}_{2^k}$ to length n and adding to this the modulus of the padded D and the code's generator polynomial, i.e.:

$$C = D \cdot 2^{n-k} \oplus \left(D \cdot 2^{n-k} \bmod \mathcal{G}\right). \tag{1}$$

Eq. 1 clearly shows that C is always a multiple of $\mathcal{G}$, since

$$\begin{aligned} C \bmod \mathcal{G} &= \left(D \cdot 2^{n-k} \oplus \left(D \cdot 2^{n-k} \bmod \mathcal{G}\right)\right) \bmod \mathcal{G} \\ &= \left(D \cdot 2^{n-k} \bmod \mathcal{G}\right) \oplus \left(D \cdot 2^{n-k} \bmod \mathcal{G}\right) \\ &= 0. \end{aligned} \tag{2}$$

By shortening the data word by m bits, the codeword will also be reduced by m bits. E.g. from $\mathcal{C}(255, 21, 55)$, one can create $\mathcal{C}(235, 1, 55)$, which has the same generator polynomial and error correction capabilities.

C. BCH decoding

BCH decoding consists of a three step process: syndrome calculation, error polynomial calculation and error position calculation. Each of these steps is explained in more detail in the next paragraphs.

1) Syndrome calculation: The first decoding step is calculating the so called *syndromes*. One takes a received codeword $R \in \mathbb{F}_{2^n}$, which is the sum of an error-free codeword C and an error vector E, and evaluates it as a polynomial. The syndromes are the evaluation results of $R(x)$ for $x = \alpha^i$, with $i = 1, \ldots, 2t$. A syndrome S_i is thus defined as

$$\begin{aligned} S_i &= R(\alpha^i) \\ &= C(\alpha^i) \oplus E(\alpha^i) \\ &= E(\alpha^i). \end{aligned} \tag{3}$$

2) Error locator polynomial calculation: Suppose we have an error vector $E = x^{l_1} + x^{l_2} + \ldots + x^{l_y}$. Then the value of the first three syndromes is:

$$\begin{aligned} S_1 &= \alpha^{l_1} + \alpha^{l_2} + \ldots + \alpha^{l_y} \\ S_2 &= \alpha^{2l_1} + \alpha^{2l_2} + \ldots + \alpha^{2l_y} \\ S_3 &= \alpha^{3l_1} + \alpha^{3l_2} + \ldots + \alpha^{3l_y} \end{aligned} \tag{4}$$

The Berlekamp-Massey (BM) algorithm [8, 9], when given a list of syndromes S_i, returns an error locator polynomial

$$\begin{aligned} \Lambda(x) &= (\alpha^{l_1}x + 1) \cdot (\alpha^{l_2}x + 1) \cdot \ldots \cdot (\alpha^{l_y}x + 1) \\ &= (x + \alpha^{-l_1}) \cdot (x + \alpha^{-l_2}) \cdot \ldots \cdot (x + \alpha^{-l_y}). \end{aligned} \tag{5}$$

One of the problems with the original BM algorithm is that it requires an inversion of an element $A \in \mathbb{F}_{2^u}$ in each of its $2t$ iterations. To eliminate this costly operation, Burton [10] devised an inversionless version of the algorithm. Multiple authors have suggested improvements to this algorithm in the form of space-time tradeoffs, e.g. [11–13].

3) Error location calculation: Finding the roots of $\Lambda(x)$ gives the location of the errors in R. The Chien search algorithm [14] is an efficient way of evaluating all possible values of α^i. It does this by improving multiplications in the evaluation formula to constant factor multiplications by noting that intermediate results for $\Lambda(\alpha^{i+1})$ differ a constant factor from intermediate results for $\Lambda(\alpha^i)$:

$$\begin{aligned} \Lambda(\alpha^i) &= \lambda_y \cdot \alpha^{it} &&+ \ldots + \lambda_1 \cdot \alpha^i &&+ \lambda_0 \\ &\equiv \lambda_{y,i} &&+ \ldots + \lambda_{1,i} &&+ \lambda_{0,i} \\ \Lambda(\alpha^{i+1}) &= \lambda_y \cdot \alpha^{(i+1)t} &&+ \ldots + \lambda_1 \cdot \alpha^{i+1} &&+ \lambda_0 \\ &= \lambda_{y,i} \cdot \alpha^t &&+ \ldots + \lambda_{1,i} \cdot \alpha &&+ \lambda_{0,i} \\ &\equiv \lambda_{y,i+1} &&+ \ldots + \lambda_{1,i+1} &&+ \lambda_{0,i+1}. \end{aligned} \tag{6}$$

III. Design

In general, BCH decoders are designed for high throughput, since they are most often used in high-throughput communication devices. In our case, however, the BCH decoder is intended for error correction of the output of a physically uncloneable function (PUF) [2], i.e. PUF key extraction [3–5]. In this setting, throughput is only a secondary requirement, since the PUF generates relatively few data to correct and error correction has to happen only once, at startup. Furthermore, a PUF key extractor is generally part of a larger design, and thus, should be as small as possible. As such, our design approach towards the BCH decoder is markedly different from the de-facto standard of using systolic arrays [11–13, 15, 16]. The primary goal of the design is to be as small as possible, and be flexible, since different PUF types require different BCH parameters. Secondary comes time efficiency. In the following section, the design of the BCH decoder in explained in detail.

A. Hardware

In order to execute the three algorithms necessary for BCH decoding a controller is needed. Furthermore, this controller needs to be easily adaptable to different code parameters, because the type of BCH code used in a PUF key extraction device depends on a lot of factors such as PUF error rate, PUF output width and final key length [4]. Due to these requirements, a microcontroller design seems best suited for the decoder design.

Components The BCH decoding processor consists of three main components, which are shown in Fig. 1. Each of them is described in the next paragraphs.

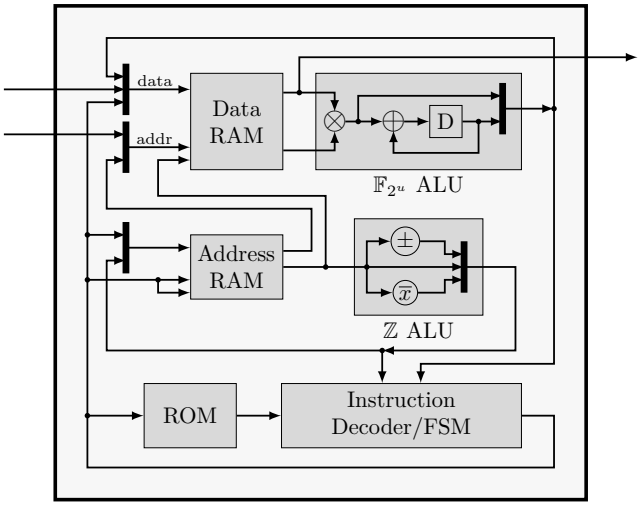

Figure 1. High-level architecture of the BCH decoder coprocessor.

1) Data block: The data block consists of a data RAM block, which stores all data necessary for the decoding as well as the corrected codeword, and an attached arithmetic unit (ALU). Since virtually all arithmetic for BCH decoding is over elements in $\mathbb{F}_{2^u}$, only a single Galois field operation is supported by the ALU: single-cycle multiply-accumulate, with the ability to execute either multiplication or addition separately. The ALU contains a single register for the accumulator and has a dual port input from the RAM.

2) Address block: Part of the novelty of our design is the use of a dedicated address block. This block consists of a tiny address RAM, of only 5 elements, and an attached ALU. The reason for including a separate address block is explained later on. The ALU works over elements in $\mathbb{Z}$ and supports increase by one, decrease by one and binary inversion, which is equal to negate and decrease by one

in two's complement notation. This allows the use of the address block both for address pointer storage, for array pointer arithmetic and for keeping track of counter values.

3) Controller: The controller consists of a firmware ROM, as well as an FSM to interpret this machine code and control the microprocessor.

Communication Not only are both the data block and the address block controlled by the controller, both also have outputs connected to it. This allows the controller to compare the content of the RAMs or the result of an arithmetic operation to some fixed value. The controller can block write signals going to both RAM blocks, which allows conditional execution for all instructions.

Code analysis on the three algorithms shows that almost every arithmetic operation takes place on array elements. This lead to the development of the address block, which allows very efficient array pointer arithmetic. The address input of the data RAM is wired straight to the output of the address RAM. Therefore only indirect access of data elements is supported. Since at most five address pointers are needed at any time, the address to these pointers can be included in each instruction word. Thus, this "forced" indirect addressing actually is one of the nice aspects of the processor, driving down both firmware size and runtime. For example, an array sum can be programmed with just three instructions: accumulate, increase address pointer and conditional branch.

B. Software

The three algorithms for BCH decoding are implemented in an assembly language for the hardware described in the previous section. In this section, we list the processor's instruction set architecture (ISA) and go over some of the techniques used to achieve a time-efficient implementation.

Instruction Set Architecture Table I lists the instruction set architecture of the processor. All instructions are 10-bits wide and contain bit fields for conditional execution and (if applicable) target and destination address pointer(s). Some instructions are implemented specifically with the target algorithms in mind. E.g. the `rotr` instruction also sets a conditional execution flag depending on the LSB of the affected data word, this eliminates the need for a separate check, allowing the implementation of the *syndrome calculation* algorithm's inner loop with only two instructions.

Optimization Techniques In order to improve the runtime of our firmware a few techniques are used.

First of all, the algorithm's inner loops are all unrolled. This reduces the overhead of costly conditional jumps back to the start of the loop. Pre- and post-loop patch code is avoided by manually tuning the number of loop unrolls to the code parameters, which keeps the impact on firmware size low. This loop optimization technique improves the runtime of our initial firmware up to 30%. The next big improvement in runtime is due to the combination of multiplication and addition in a single-cycle instruction. The merge of these two instructions results in a further 38% speedup of our error location calculation algorithm. Code duplication, in order to move conditional branches out of loops, improves the runtime of the Berlekamp-Massey implementation by another 20%. The support for conditional execution speeds up the syndrome calculation algorithm further, by 28%, due to the elimination of conditional jumps in the inner loop. Finally, the last improvement to runtime is due to improved memory management, with syndrome calculation seeing a 64% speed increase over an implementation with straightforward variable placement.

Table I
Instruction set architecture of the processor.

Opcode	Result	Cycles
`jump`	PC $\leftarrow$ *value*	2
`cmp_jump`	PC $\leftarrow$ *value* **if** (*comp* = **true**)	3
`stop`	PC $\leftarrow$ PC	1
`comp`	$\text{cond}_i \leftarrow$ (*comp* = **true**)	2
`set_cond`	$\text{cond}_i \leftarrow$ *value*	1
`load_reg`	reg $\leftarrow \text{data}[\text{addr}_i]$	1
`load_fixed_reg`	reg $\leftarrow$ *value*	2
`load_fixed_addr`	$\text{addr}_i \leftarrow$ *value*	2
`mod_addr`	$\text{addr}_i \leftarrow f(\text{addr}_i)$	1
`copy_addr`	$\text{addr}_i \leftarrow \text{addr}_j$	1
`store_reg`	$\text{data}[\text{addr}_i] \leftarrow$ reg	1
`store_fixed`	$\text{data}[\text{addr}_i] \leftarrow$ *value*	2
`rotr`	$\text{data}[\text{addr}_i] \leftarrow \text{data}[\text{addr}_i] \circlearrowleft 1$	1
`shiftl_clr`	$\text{data}[\text{addr}_i] \leftarrow \text{data}[\text{addr}_i] \ll 1$	1
`shiftl_set`	$\text{data}[\text{addr}_i] \leftarrow (\text{data}[\text{addr}_i] \ll 1) \mid 1$	1
`gf2_add_mult`	$\text{data}[\text{addr}_i] \leftarrow \text{data}[\text{addr}_i] \otimes \text{data}[\text{addr}_j]$ reg $\leftarrow$ reg $\oplus$ ($\text{data}[\text{addr}_i] \otimes \text{data}[\text{addr}_j]$)	1

IV. Implementation

In the next paragraphs, we list the results for FPGA synthesis of our design and show the impact of code parameters on runtime.

Synthesis Our design is completely implemented in Verilog and was synthesized for the Xilinx© Virtex-6™ family of FPGAs using Xilinx ISE 12.2 M.63c with design strategy 'Area reduction with Physical synthesis'. As can be seen in Table II, the total size of our design is very small and changes little for different BCH codes. No separate RAM blocks are used, since our design uses RAM & ROM blocks which are implemented within LUTs. Thus, the listed slice count is the actual total size that the design requires.

Table II
Synthesis results for implementation on a Xilinx© Virtex-6™.

BCH(n, k, t)	Area [slice]	Area [FF]	Area [LUT]	F_{max} [MHz]
$413, 296, 13$	65	33	244	94.4
$380, 308, 8$	66	33	244	97.8
$318, 174, 17$	68	33	251	93.6

To the best of our knowledge, a comparison with existing BCH decoders is near impossible and makes little sense. This is due to the target application of our design: PUF key extraction. The primary goal of our design is compactness, for existing designs it is high throughput [11–13, 15–17]. Further complicating this is that the area of other implementations are either given for an ASIC implementation [11, 15, 16] or simply not stated [12, 13, 17]. Furthermore, the codes used for our target application are generally defined over $\mathbb{F}_{2^u}$ where $8 \leq u \leq 10$, with high error correcting capabilities of 3–10% [3–5], and our firmware is optimized with this in mind. We have not been able to find designs for such code parameters. Finally, most publications deal with Reed-Solomon decoding, which requires slightly different algorithms than those needed for BCH decoding, making fair comparisons even harder.

Runtime Code parameters greatly influence the runtime of each algorithm. The high-order approximate formulas for each algorithm's runtime in Table III clearly show that t has the largest influence, unless very long BCH codes are used. In this same table, formulas are given for the *ideal* runtime, which we define as: the number of cycles needed if each inner loop iteration takes one cycle, no matter how many operations are inside the loop, without parallel execution.

Comparing these *ideal* runtime formulas with the formulas for our implementation shows that the coprocessor is very efficient. Of note are the syndrome calculation and error location calculation implementations, which on average require only 2–4 times more cycles than in the *ideal* case, even with the overhead of conditional loop branches.

Table III
High-order approximations for algorithm runtime. *Ideal* assumes single cycle inner loops, no parallelism.

Algorithm	**Runtime [cycles]**	
	Ideal	**Actual**
Syndrome calculation	$2t \cdot n$	$40t \cdot \lceil \frac{n}{u} \rceil$
Berlekamp-Massey	$3.5 \cdot (t^2 + t)$	$36t^2$
Error loc. calculation	$t \cdot n$	$3.6t \cdot n$

Table IV lists the runtime of our processor for the example BCH codes. It clearly shows that the number of corrigible errors t has the largest effect on the runtime.

Table IV
Actual number of cycles required for BCH decoding.

BCH(n, k, t)	**Runtime [cycles]**
$413, 296, 13$	55 379
$380, 308, 8$	26 165
$318, 174, 17$	50 320

V. Conclusion

We have presented the design and implementation of both hard- and software for a tiny application-specific programmable BCH decoding processor. Our design requires less than 1% (70 slices) for a BCH(413, 296, 13) decoder on a small Virtex-6 FPGA, and gets close to the *ideal* runtime for two out of three required algorithms.

Due to its extremely small size, it is the perfect match for a PUF key extraction system. Such a system will spend multiple milliseconds interfacing a PUF [4] and thus the speed of our design is well within acceptable limits.

Acknowledgments

This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007) and by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. In addition, it was supported by the Flemish Government, FWO G.0550.12N and by the European Commission through the ICT programme under contract FP7-ICT-2011-284833 PUFFIN and FP7-ICT-2007-238811 UNIQUE.

Bibliography

[1] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, "Ron was wrong, Whit is right," Cryptology ePrint Archive, Report 2012/064, 2012, http://eprint.iacr.org/.

[2] R. Maes and I. Verbauwhede, "Physically Unclonable Functions: A Study on the State of the Art and Future Research Directions," in *Towards Hardware-Intrinsic Security*, ser. Information Security and Cryptography, A.-R. Sadeghi and D. Naccache, Eds. Springer Berlin Heidelberg, 2010, pp. 3–37.

[3] R. Maes, R. Peeters, A. Van Herrewege, C. Wachsmann, S. Katzenbeisser, A.-R. Sadeghi, and I. Verbauwhede, "Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-enabled RFIDs," in *Lecture Notes in Computer Science*. Springer-Verlag, Feb. 2012.

[4] R. Maes, A. Van Herrewege, and I. Verbauwhede, "PUFKY: A Fully Functional PUF-based Cryptographic Key Generator," *CHES 2012*, 2012, in press.

[5] C. Bösch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls, "Efficient Helper Data Key Extractor on FPGAs," in *CHES*, ser. Lecture Notes in Computer Science, E. Oswald and P. Rohatgi, Eds., vol. 5154. Springer, 2008, pp. 181–197.

[6] R. C. Bose and D. K. Ray-Chaudhuri, "On a Class of Error Correcting Binary Group Codes," *Information and Control*, vol. 3, no. 1, pp. 68–79, Mar. 1960.

[7] A. Hocquenghem, "Codes Correcteurs d'Erreurs," *Chiffres*, vol. 2, pp. 147–156, Sep. 1959.

[8] E. Berlekamp, "On Decoding Binary Bose-Chadhuri-Hocquenghem Codes," *IEEE Transactions on Information Theory*, vol. 11, no. 4, pp. 577–579, Oct. 1965.

[9] J. Massey, "Shift-Register Synthesis and BCH Decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, Jan. 1969.

[10] H. Burton, "Inversionless Decoding of Binary BCH codes," *IEEE Transactions on Information Theory*, vol. 17, no. 4, pp. 464–466, Jul. 1971.

[11] J.-I. Park, K. Lee, C.-S. Choi, and H. Lee, "High-Speed Low-Complexity Reed-Solomon Decoder using Pipelined Berlekamp-Massey Algorithm," in *2009 International SoC Design Conference (ISOCC)*, Nov. 2009, pp. 452–455.

[12] I. Reed and M. Shih, "VLSI Design of Inverse-Free Berlekamp-Massey Algorithm," *IEEE Proceedings on Computers and Digital Techniques*, vol. 138, no. 5, pp. 295–298, Sep. 1991.

[13] D. Sarwate and N. Shanbhag, "High-Speed Architectures for Reed-Solomon Decoders," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 5, pp. 641–655, Oct. 2001.

[14] R. Chien, "Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes," *IEEE Transactions on Information Theory*, vol. 10, no. 4, pp. 357 – 363, Oct. 1964.

[15] W. Liu, J. Rho, and W. Sung, "Low-Power High-Throughput BCH Error Correction VLSI Design for Multi-Level Cell NAND Flash Memories," in *SiPS*. IEEE, 2006, pp. 303–308.

[16] J.-I. Park, H. Lee, and S. Lee, "An Area-Efficient Truncated Inversionless Berlekamp-Massey Architecture for Reed-Solomon Decoders," in *2011 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2011, pp. 2693–2696.

[17] H.-C. Chang and C. Shung, "New Serial Architecture for the Berlekamp-Massey Algorithm," *Communications, IEEE Transactions on*, vol. 47, no. 4, pp. 481–483, Apr. 1999.